OULUN AMMATTIKORKEAKOULU

Oula Helander

DEVELOPMENT OF STATUS MONITORING AND ERROR HANDLING FRAMEWORK FOR NEMO FIRMWARE MANAGER

DEVELOPMENT OF STATUS MONITORING AND ERROR HANDLING FRAMEWORK FOR NEMO FIRMWARE MANAGER

Oula Helander
Bachelor's Thesis
Spring 2018
Information and Communication Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information and Communication Technology, Software Development

---

Author: Oula Helander
Title of thesis: Development of Status Monitoring and Error Handling Framework for Nemo Firmware Manager
Supervisor: Pekka Alaluukas
Term and year when the thesis was submitted: Spring 2018
Pages: 45

This thesis was made for Keysight Technologies. The framework was made to the Nemo Firmware Manager which is used to update custom firmware to devices. The Nemo Firmware Manager did not have any kind of logging system in case there was a problem occurring when running the program.

There was a need for a system that logs important event data when the Nemo Firmware Manager is running. The framework is meant to be an independent task which does not interfere with the normal usage of the Nemo Firmware Manager. Once the framework is operational, the data will be synchronized to the cloud database and the data is retrievable by the admin user.

The project utilized design patterns, object-oriented and asynchronous programming. Since the Nemo Firmware Manager is a software program which has been in development for years, the project also utilized the libraries which had been implemented previously. These methods were essential for a program that is supposed to run with modular code. The Nemo Firmware Manager is being developed on cross-platform IDE, Xamarin for Visual Studio, and is currently having the Android and the Windows platform being developed.

In the future, the framework will continue to be developed as new Nemo Firmware Manager -features come up. This thesis sums up all the core components of the framework and explains the how acomponents work.

Keywords: Nemo Firmware Manager, logging system, design patterns, object-oriented, asynchronous, programming, modular code, cross-platform, Xamarin, core components

PREFACE

I would like to thank Keysight Technologies for making it possible to do the necessary project work and thesis for them.

Special thanks to my mentor inside the company, senior software engineer Heikki Siikaluoma, for guiding me during the making of this framework as well as to any of the company colleagues who helped in any way with this work.

Oulu, 5.4.2018

Oula Helander

TABLE OF CONTENTS

VOCABULARY

API

Application Programming Interface

C4-model

Guideline to represent software architecture in clear way. The C's come from context, container, components and code

C#

Programming language used to code this framework project

DLL

Dynamic-Link Library

FIFO buffer

First In, First Out buffer method

GUI

Graphical User Interface

HTTPS

Hypertext Transfer Protocol Secure

IDE

Integrated Development Environment

NFM

Nemo Firmware Manager program

OOAD

Object-Oriented Analysis and Design

OOP

Object-Oriented Programming

Software Design Pattern

Reusable solutions to common problems in programming

TFS

Team Foundation Server -tool

TLS

Transport Layer Security

UML-diagram

Unified Model Language -diagram

Xamarin

Allows to write native Android, iOS and Windows cross-platform applications

1 INTRODUCTION

The thesis was made for Keysight Technologies. Keysight Technologies is a global multicultural company with over 10,000 employees all over the world. Specifically, the project was made to Nemo Wireless Network Solutions that provides network testing and analyzing solutions to its customers. The framework is part of the application called the Nemo Firmware Manager.

The Nemo Firmware Manager is software that is used to install a company's custom-built firmware to smartphones. The software has been under development for many years and already has multiple iterations released for customer use. The software is currently developed using Visual Studio with Xamarin. Xamarin allows easier development to multiple platforms by offering single programming language and a cross-platform libraries utility. These libraries can be used on Android, iOS, and Windows -platforms making the development of the same functionalities to each platform much faster and smoother. The Nemo Firmware Manager software currently utilizes Windows and Android platforms.

The motivation to make the framework came from the company's need to have a status monitoring and error handling system to the NFM. The Nemo Firmware Manager software had no real error logging system in place. This made problem debugging frustrating and slow for the Keysight employees. The software would give a vague error message and debugging of the root cause of the problem would be lengthy process, also requiring interaction with the customer. With the system in place, the employee - customer interaction is minimal in solving problems related to the NFM.

One of the goals of this project was to make the framework function in such a way that it would not interfere with the NFM software. The framework would simply pick up data while the NFM was running and send it to the server if there was a connection to the server. If there was no connection, the data would be stored in a cache and in a local file.

2 DEVELOPMENT

The Nemo Firmware Manager has been under development for many years before the thesis work and multiple iterations have already been released. This meant that the libraries implemented to the NFM were comprehensive and this work used some of these libraries. The work was done using Visual Studio with Xamarin as the IDE in the development of this project and utilized the cross-platform capabilities which the environment provided. The goal was to make the framework as modular as possible by using the cross-platform capability.

We used Hungarian writing notation for the code. It eases the readability of the code because every pointer and member are notated with a set of rules. For example, m_iCounter where the “m” means member of a structure or class and the “i” refers to the type of the member. With Visual Studio’s Intellisense it is easy to sift through the members when Hungarian notation is used (1).

2.1 Nemo Firmware Manager structure benefits

Figure 1 shows the architecture of the Nemo Firmware Manager. Here we can see how the cross-platform capability is used and what DLLs are included in which platforms. Currently, the NFM has Android and Windows platforms being developed and the NFM is taking advantage of the cross-platform functionality by using the “NemoCrossPlatform” library.

This framework has been implemented in Microsoft Windows Desktop (C#) -platform in the “NemoCore”, but the framework has been developed to be as modular as possible and much of the functionality is in the cross-platform library called “NemoCrossPlatform”. Every class of the framework in the “NemoCore” inherits from “NemoCrossPlatform” classes and most of the classes in “NemoCrossPlatform” are abstract classes. This makes the implementation of the framework into another platform feasible and relatively easy.

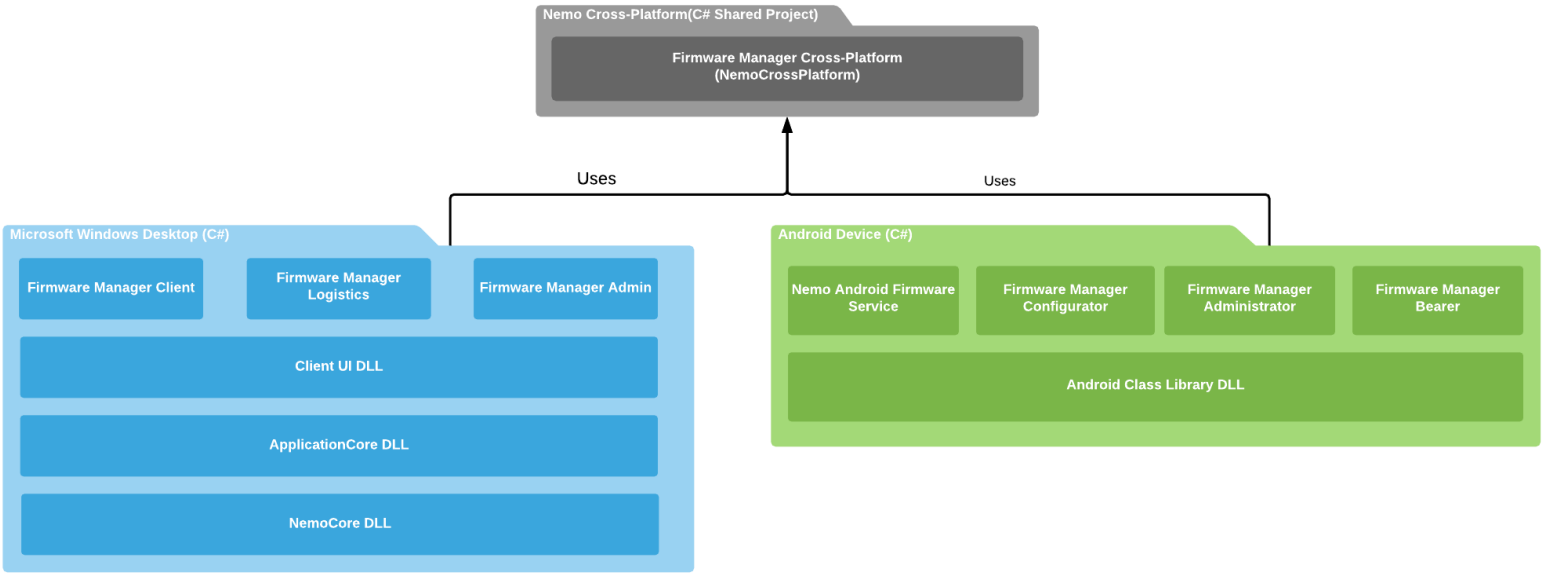

FIGURE 1. Architecture of Nemo Firmware Manager

2.2 Version control

We used a version control tool called the Team Foundation Server. Version control was needed because the NFM has multiple developers. A separate branch was created as a research branch for the work. Usually, the latest version of the framework was pushed to the TFS before the weekly meeting. Once the framework is ready for release, the framework's code will be integrated into the main branch of the NFM.

2.3 Design patterns

Design patterns are used as templates for solving general repeatable problems. Patterns are not complete designs which can be transferred to the code directly. Design patterns allow the developers to communicate well-known and well understood names for the implementations. The framework utilizes some of these design patterns such as factory, event and singleton patterns (2).

In the factory pattern, the factory completely abstracts the creation and initialization of the product from the client. This indirection enables the client to focus on its discrete role in the application without concerning itself with the details of how

the product is created. Thus, as the product implementation changes over time, the client remains unchanged (3, 4).

The singleton pattern makes sure that only one instance of the class is initialized and provides a global access point to it. The singletons used in this framework are all done with lazy initialization, meaning that the creation of the instance is done on first use (5).

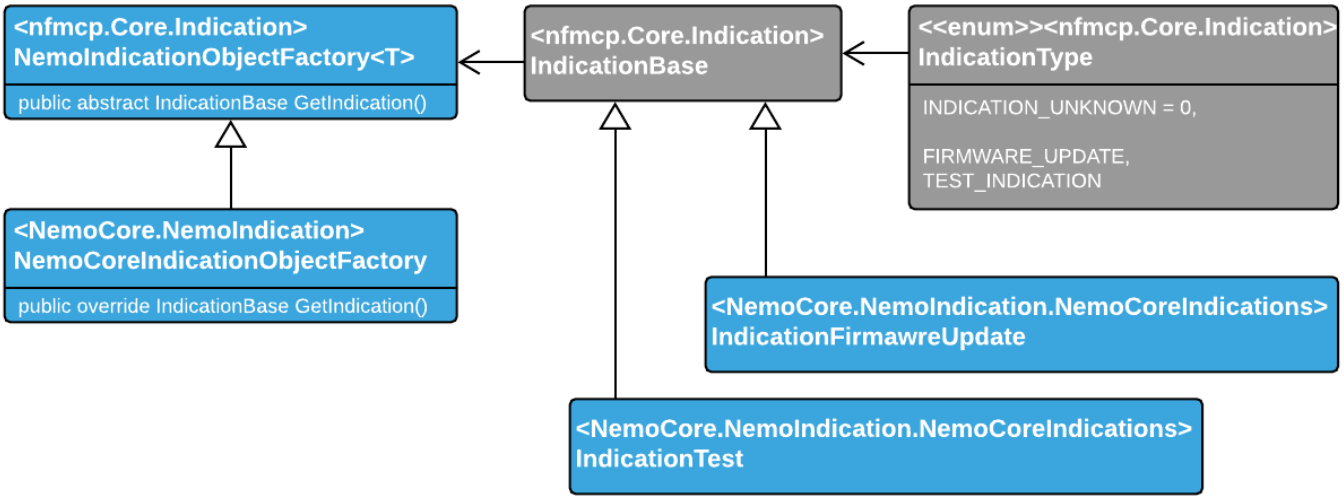

FIGURE 2. Factory pattern and singleton pattern used in the framework

Figure 2 shows how the factory pattern has been implemented to the work. Here the abstract classes are located in the cross-platform library (nfmcp) of the NFM project and the class is inherited to “NemoCore” which is in the Windows platform section. In “NemoCore” the inherited classes are implemented, and the necessary methods have been overridden to add their Windows platform specific functionality.

Overall, the implementation of the design patterns to the framework was feasible and easy. The design patterns are in common use with developers and this also means they are well documented which makes them easy to take in use.

2.4 Agile software development

The development of this framework was made by using the agile software development methods. The iteration cycle was 1 week long and when the iteration was finished, we had meeting with the senior software engineer. In these meetings we went through the last weeks progression and agreed what the new iteration should contain. We utilized several UML-diagrams to communicate the programmed code.

After every week I kept a work diary. I wrote down all the things I got done and I also listed the main problems on that specific week. I also made small to do -list for the next weeks iteration and it was helpful for remembering the tasks.

3 DOCUMENTATION

The tools used to document the project were LucidChart -web client, Notepad++, Microsoft Visio, Word, and Power Point. The main tool I used to make the UML-diagrams was Lucidchart -web client. The framework system quickly became hard to grasp. It was necessary to visualize the logic between classes by using UML-diagrams

The UML-diagrams we used were class, sequence and use case diagrams. These diagrams were necessary for the weekly meetings to plan up the next iteration and represent the work that had been done. We also started using C4-models which were helpful in representing the whole system.

3.1 Use case diagrams

At the start of the project, I made use case diagrams from the most common problems. These are seen in figures 1, 2, and 3. With these scenarios, I could also make a “general pattern for solving the NFM problems” diagram which is seen in figure 4. These use case diagrams helped to determine what kind of implementation for the framework would be best. Based on these use cases we ended up making the system as independent from customer interaction as possible. This smoothens and quickens the process that the developer and technical support must do to debug the problem.

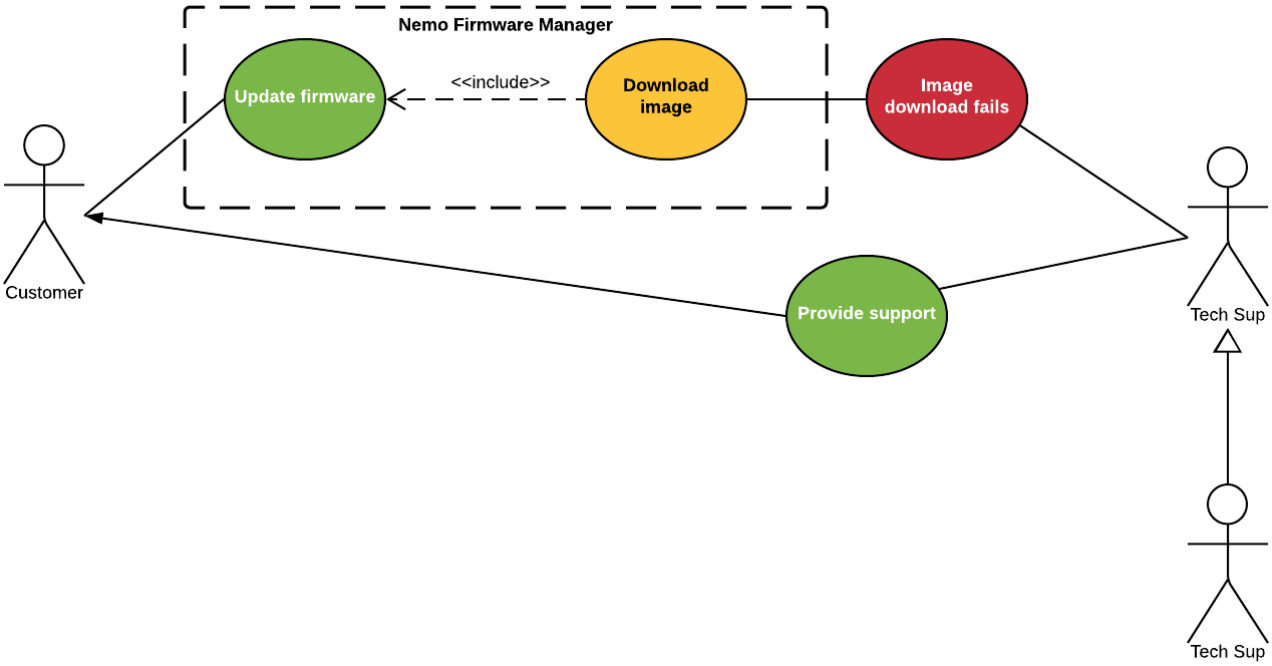

FIGURE 3. Use case diagram of possible scenario 1

In Figure 3, the customer attempts to flash the phone and the NFM checks if the correct firmware is on the local drive. If the image is not on the drive the NFM attempts to download it. If the image download fails for some reason error message is displayed and the customer needs to contact the technical support.

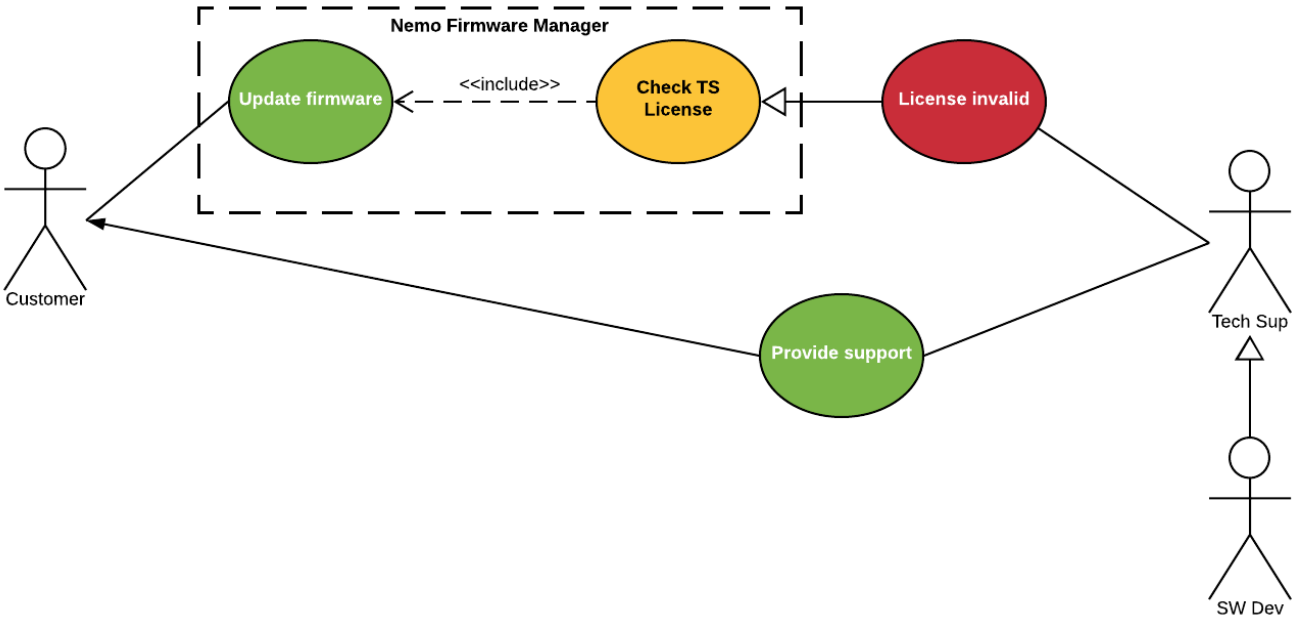

FIGURE 4. Use case diagram of possible scenario 2

In figure 4, the customer attempts to flash the phone and the NFM then checks the technical support date of the license. If the license is invalid the customer must create a ticket regarding this error. The technical support team will then assist with the issue.

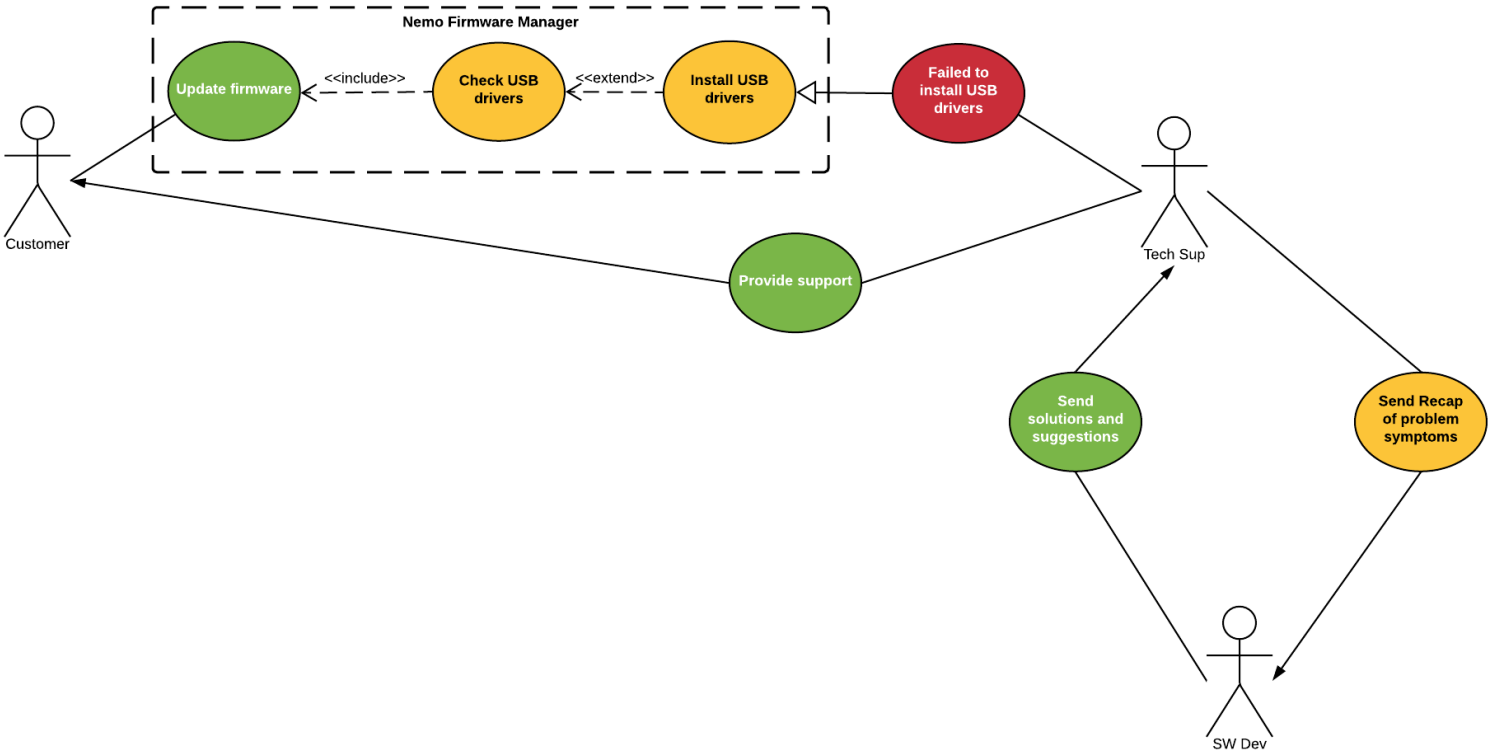

FIGURE 5. Use case diagram of possible scenario 3

In figure 5, the customer attempts to flash the phone and the NFM checks that the right USB drivers are installed. If the NFM does not find the right drivers, the program will try to install the drivers. If the installation fails for some reason, the error message is shown. This results in a ticket being sent to the technical support who will then assist the customer with the installation.

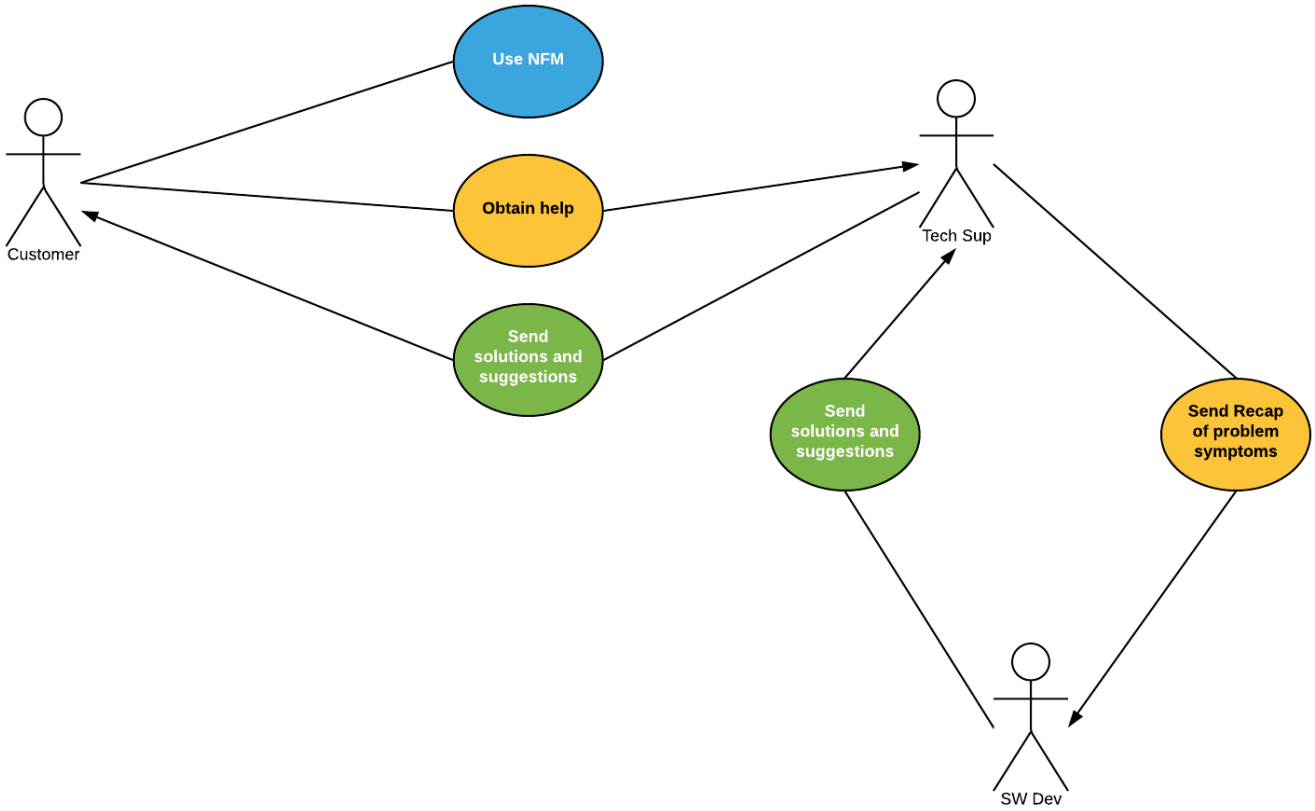

FIGURE 6. Use case diagram of general pattern

In figure 6, the customer has a problem with the NFM and contacts the technical support team. The technical support member collects data from the customer about the error and sends them to software developer. The software developer makes a list of suggestions and possible solutions and sends them to a technical support member, who then redirects those suggestions and solutions to the customer.

3.2 C4-Models

C4-models use a different kind of method to represent the system. The idea is to “zoom in” to the system specifics gradually. This helps to communicate the entirety of how the system works. The four Cs come from context, container, component and the code itself. Overall, I found the usage of these C4-models helpful and I see why these types of diagrams are needed. These diagrams make it easy to grasp the whole system (6).

Below are the C4-model diagrams of the indication framework in a Windows platform. Firstly, in figure 7 it possible to see how the framework fits in to the NFM. In figure 8, the diagram shows the libraries somewhat separated. It also shows major classes and events shown. Lastly, in figure 9 shows the inside of indication framework. We can see the components and their relations to each other as well as the flow of how the indication data goes through the framework.

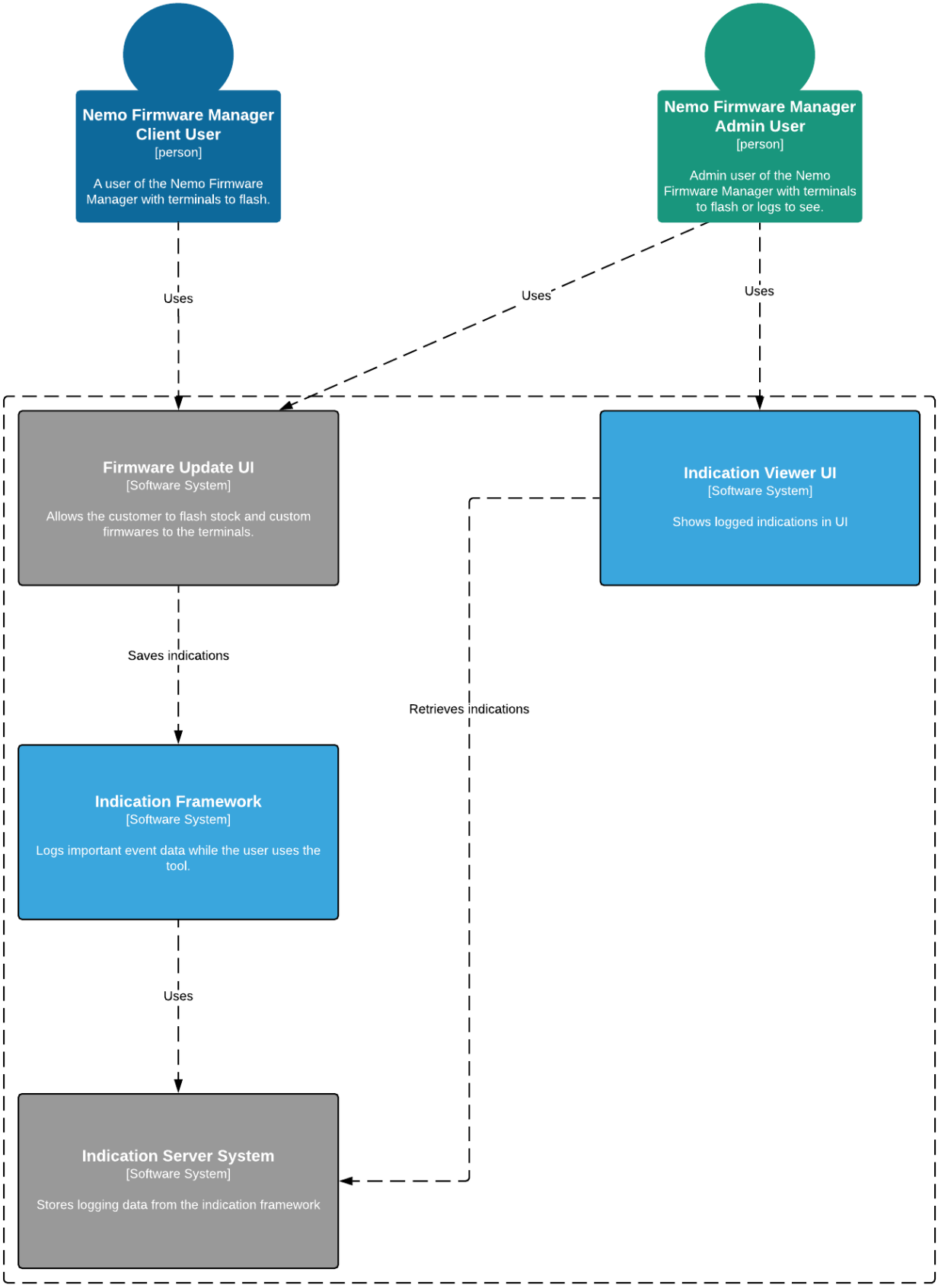

FIGURE 7. Framework's system context presented as C4-model context diagram

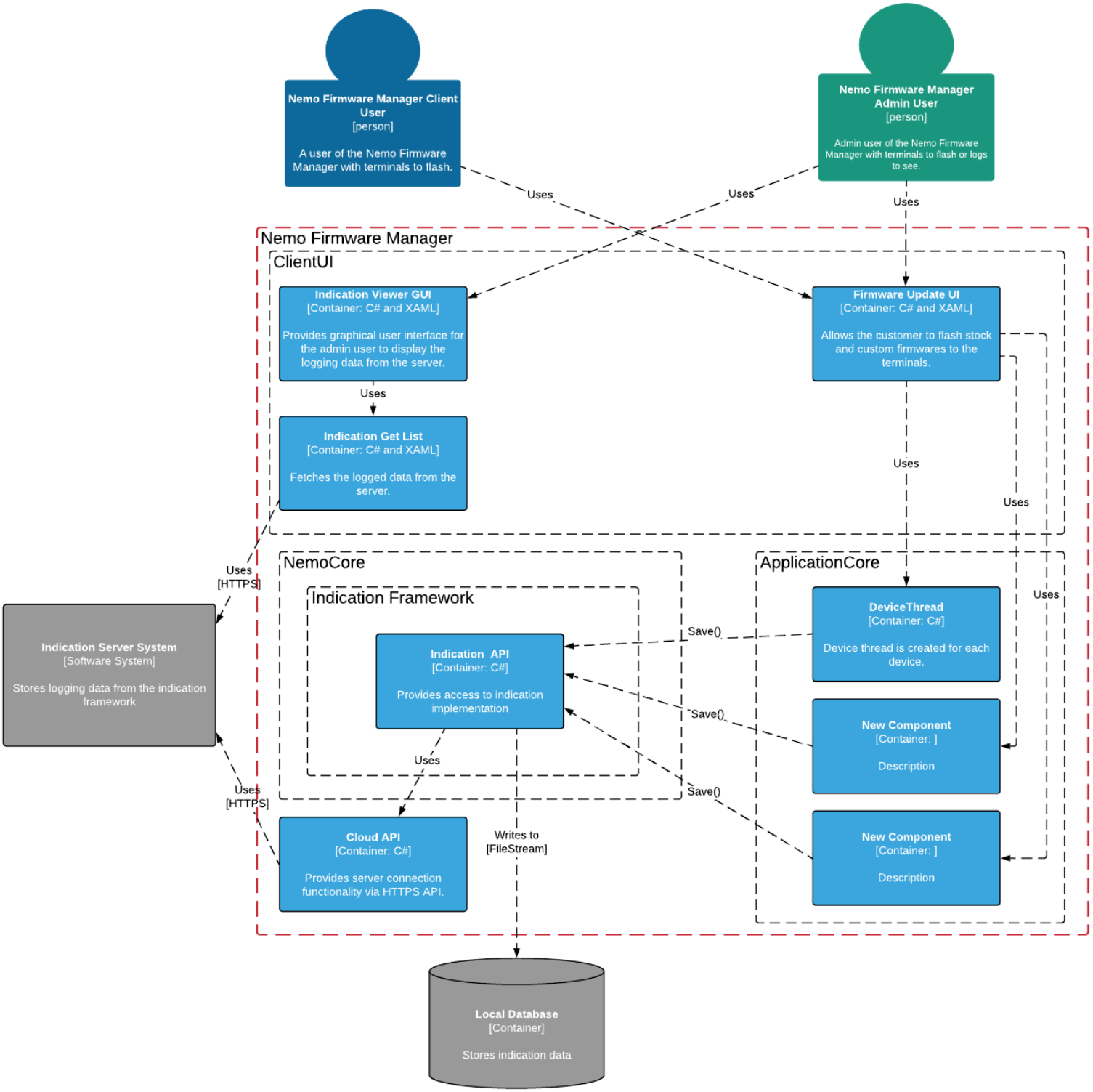

FIGURE 8. Framework presented as C4-model container diagram

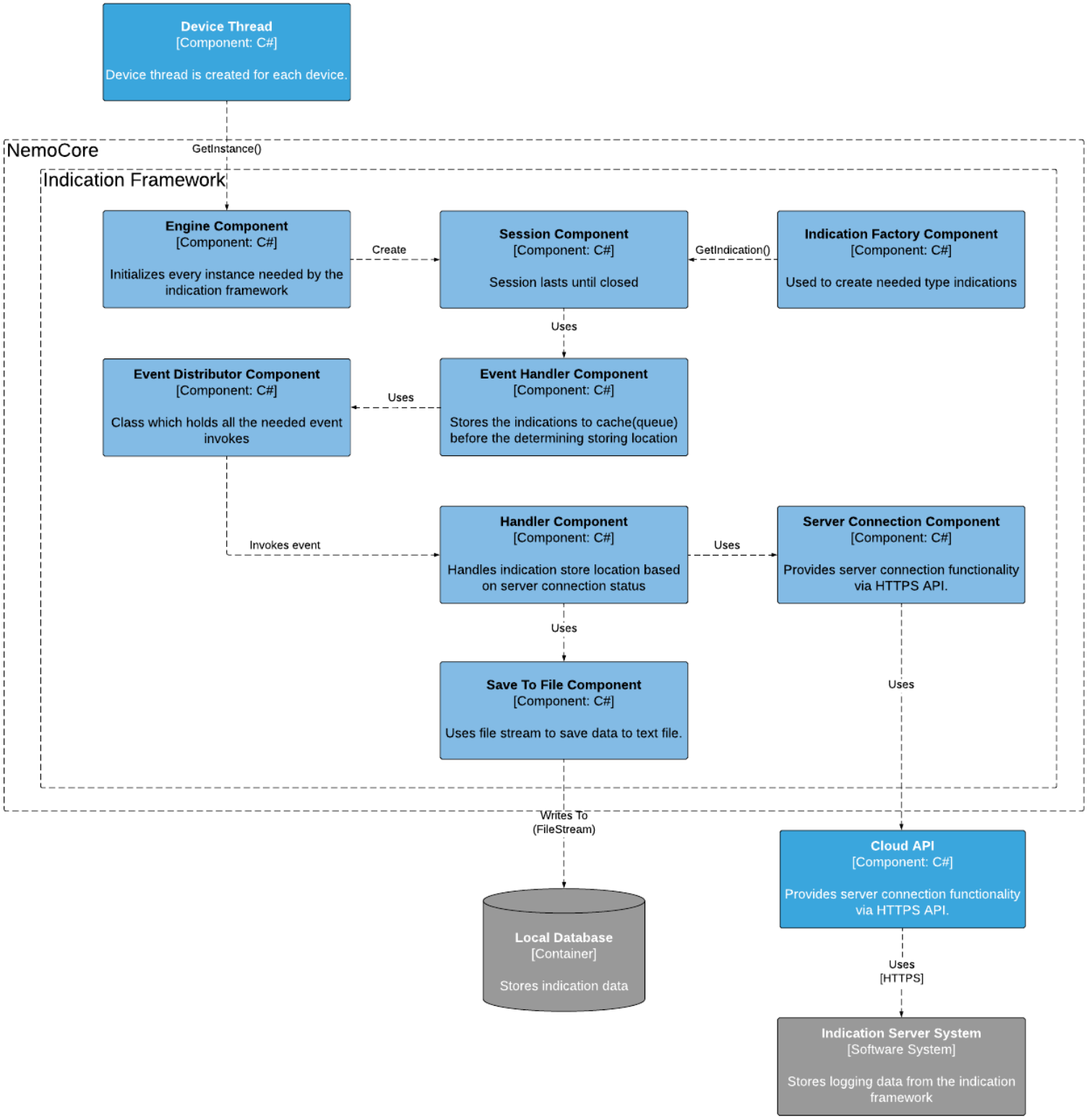

FIGURE 9. Framework presented as C4-model component diagram

3.3 Class diagrams

The framework's classes are represented as class diagrams. From these diagrams you can see the relations of the classes. Every class that has been inherited is in the "NemoCore" and every parent class is in the "NemoCrossPlatform". This style of architecture allowed for as modular a code as possible. All the class diagrams represented in chapter 4 end up in the engine component which initializes all the other necessary components.

3.4 Sequence diagrams

The sequence diagrams were essential in our weekly meetings to show my progress and to show the logic behind every component. In the best-case scenario, making a sequence diagram of the event helps you see better solutions for that event. Every major event in the framework has its own sequence diagram, as depicted in chapter 4.

4 THE INDICATION FRAMEWORK

This chapter presents the class and sequence diagrams of each component set and explains the key elements in them. All these classes end up in the Nemo-CoreIndicationEngine class which initializes all the components needed by the new session. The engine and session classes wrap up the whole framework into an easily accessed API.

The framework's data saving logic revolves around the status of the server connection. If the connection is not found, the framework will start caching the data and saving it to a file. If the connection to the server is established, the framework will start dumping the cached data to the server while simultaneously taking into account the new indication data coming from the NFM.

You can also see that the initializations are within a synchronization lock. This prevents multiple threads from accessing the resources at the same time when a number of terminals create their sessions at once (7).

4.1 Engine and session component

One of the important goals was to make the framework easily accessible and therefore make it easy to add this logging feature to the existing code. To initialize a new logging session, we must call the NemoCoreIndicationEngine singleton instance. The engine will initialize every component needed by the new session if they are not already initialized. The NemoCoreIndicationEngine instance has method called Start(). Calling it will return NemoCoreIndicationSession as shown in figure 10. In figure 11, we can also see the sequence diagram for the Start() method. Each terminal connected to the NFM creates its own session.

```
public NemoCoreIndicationSession Start(IDeviceNemoCloud pTargetDevice)
{
    if (NemoCoreDebugManager.bNemoCoreIndicationEngine) NemoCoreDebugManager.LogDebug("NemoCoreIndicationEngine: Start...");

    lock (m_pSynclock)
    {
        IndicationEventHandler pIndicationEventHandler = CreateIndicationEventHandler(m_pNemoCoreEventDistributor);

        NemoCoreIndicationSession pNemoCoreIndicationSession = new NemoCoreIndicationSession(pTargetDevice, pIndicationEventHandler);

        if (NemoCoreDebugManager.bNemoCoreIndicationEngine) NemoCoreDebugManager.LogDebug("NemoCoreIndicationEngine: Start completed");

        return pNemoCoreIndicationSession;
    }
}
```

FIGURE 10. The Start() method in the NemoCoreIndicationEngine class

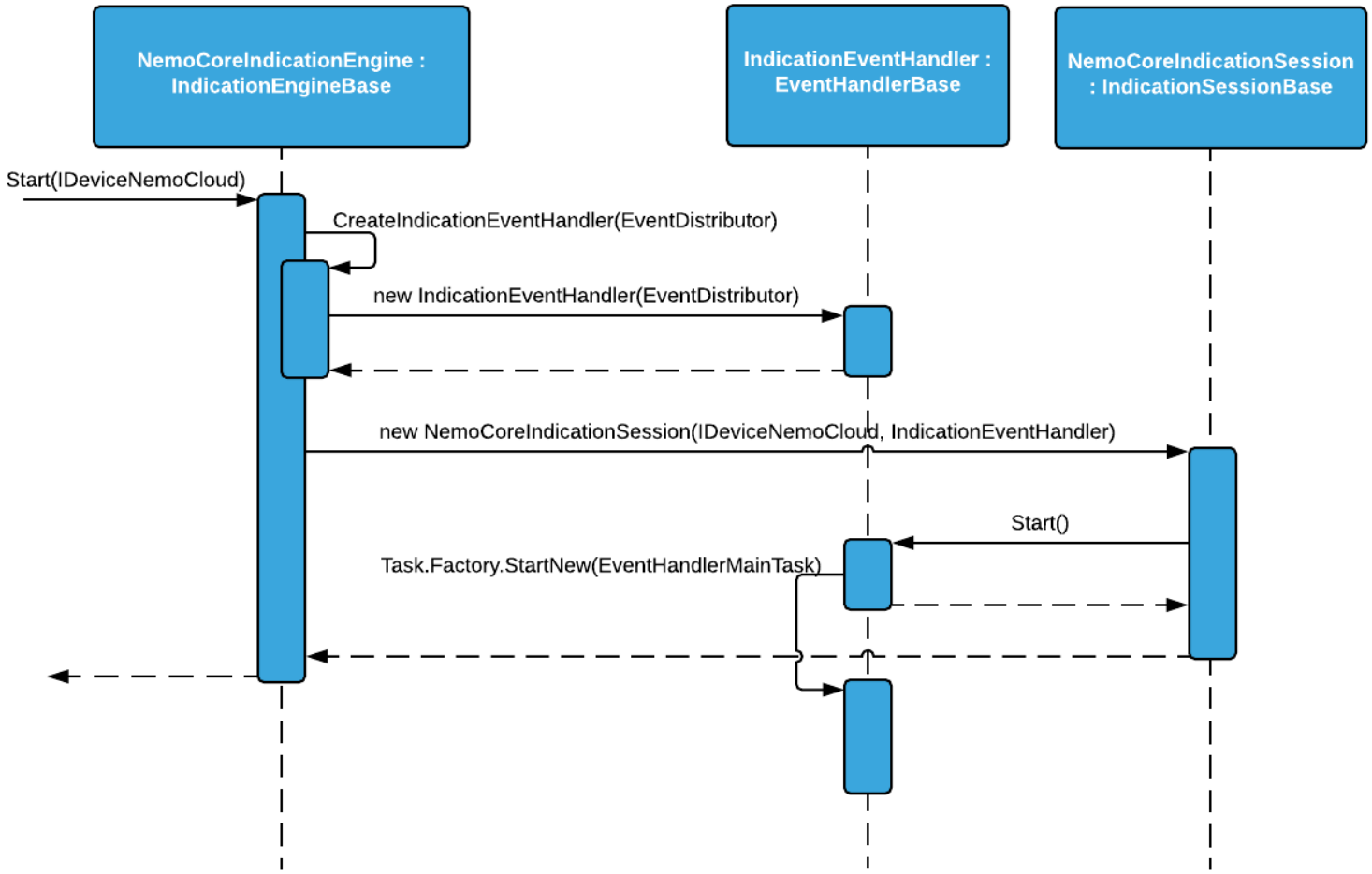

FIGURE 11. The sequence diagram from the initialization of new session

To log the wanted data, we will call the Save() method. The Save() method has parameters which determine the indication type and the logged data in string format. The indication type determines what template of indication class we should use in each logged session. You can see the API in figure 12 and the sequence diagram for the logic of Save() method in figure 13. When the logging session is completed, we will call the Close() method which will terminate the session once the cache has been emptied.

```
NemoCoreIndicationSession pNemoCoreIndicationSession = NemoCoreIndicationEngine.GetInstance().Start(m_pDevice);

pNemoCoreIndicationSession.Save(nfmcp.Core.Indication.IndicationType.FIRMWARE_UPDATE, "String to test the framework");

...

pNemoCoreIndicationSession.Close();
```

FIGURE 12. The easily accessed API of the framework to log data

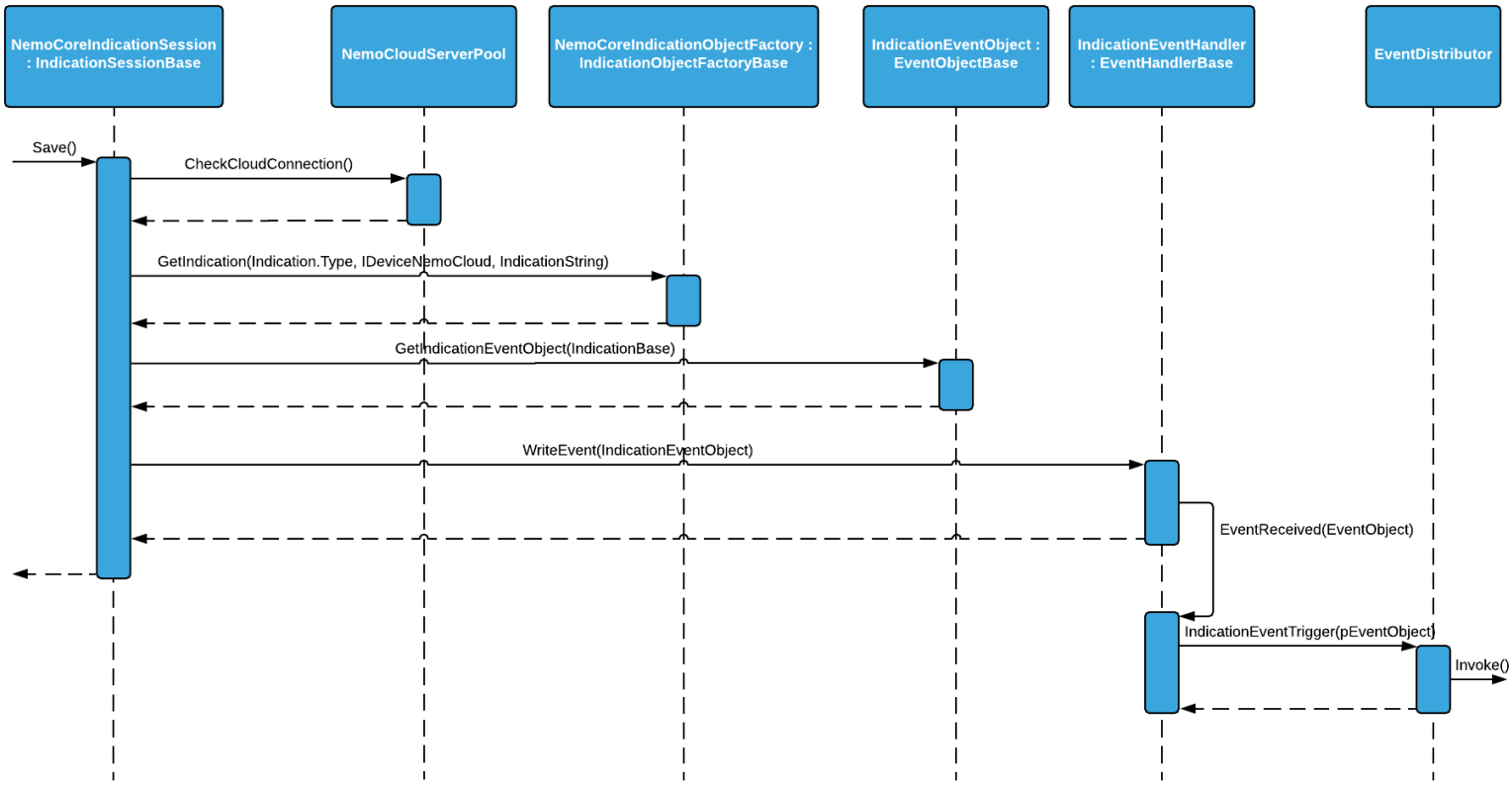

FIGURE 13. The sequence diagram of session's Save() method

Currently, there are two indication classes implemented. The IndicationTest and the IndicationFirmwareUpdate both inherit from the IndicationBase class as shown in figure 14. The number of these classes will increase in future development as new targets for logging are decided.

In figure 14, you can also see the overall structure of the components. The abstract base class is in the "NemoCrossPlatform" (nfmcp) and the inheriting class is in "NemoCore".

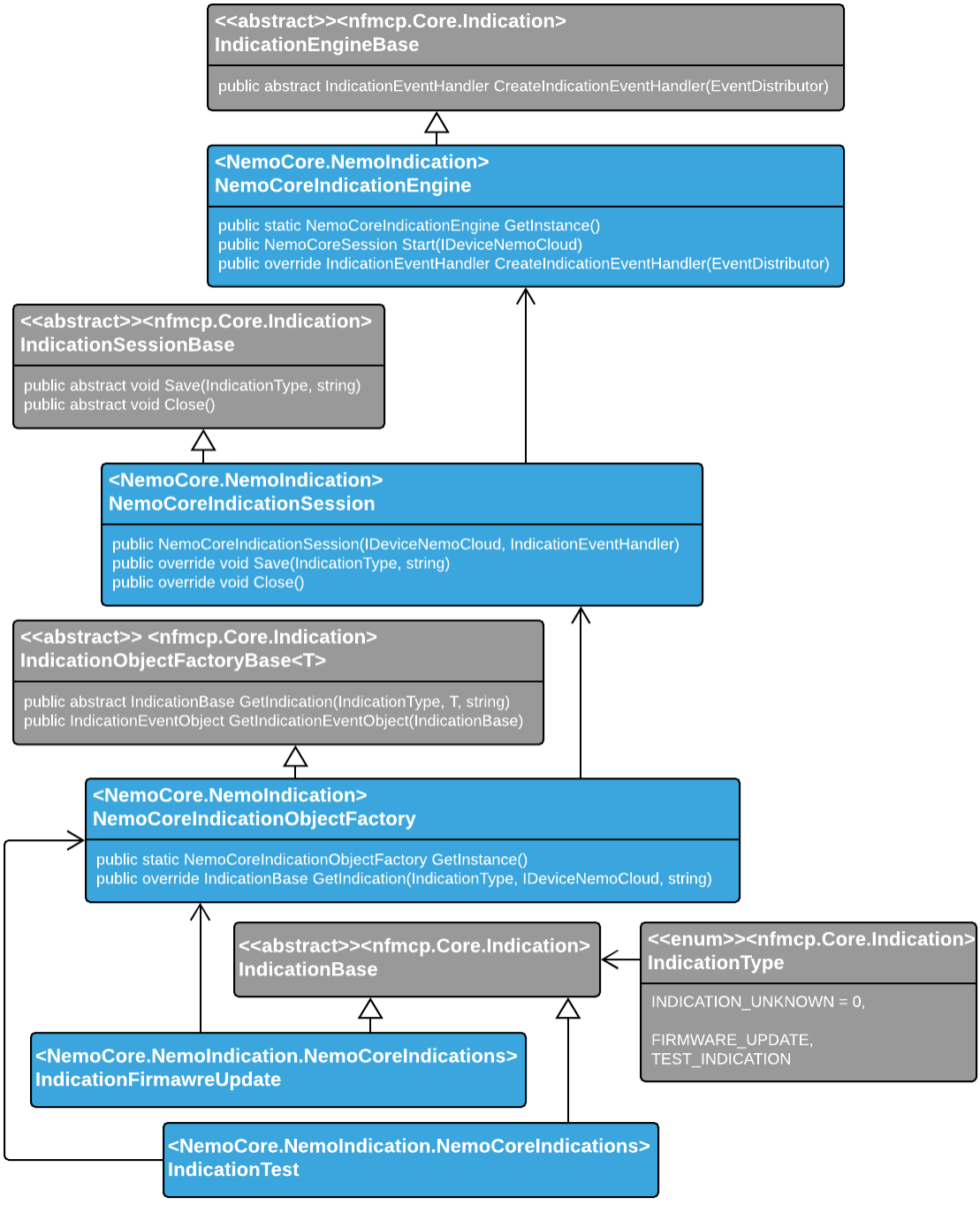

FIGURE 14. Class diagram from the session and factory implementation

4.2 Event distributor component

The event distributor component handles all the events in the framework. This class follows the standard Microsoft .NET event pattern design. Shown in figure 15, all the event declarations and invokes are centralized under one class. This enables that new events are easy to add to the EventDistributor class (8).

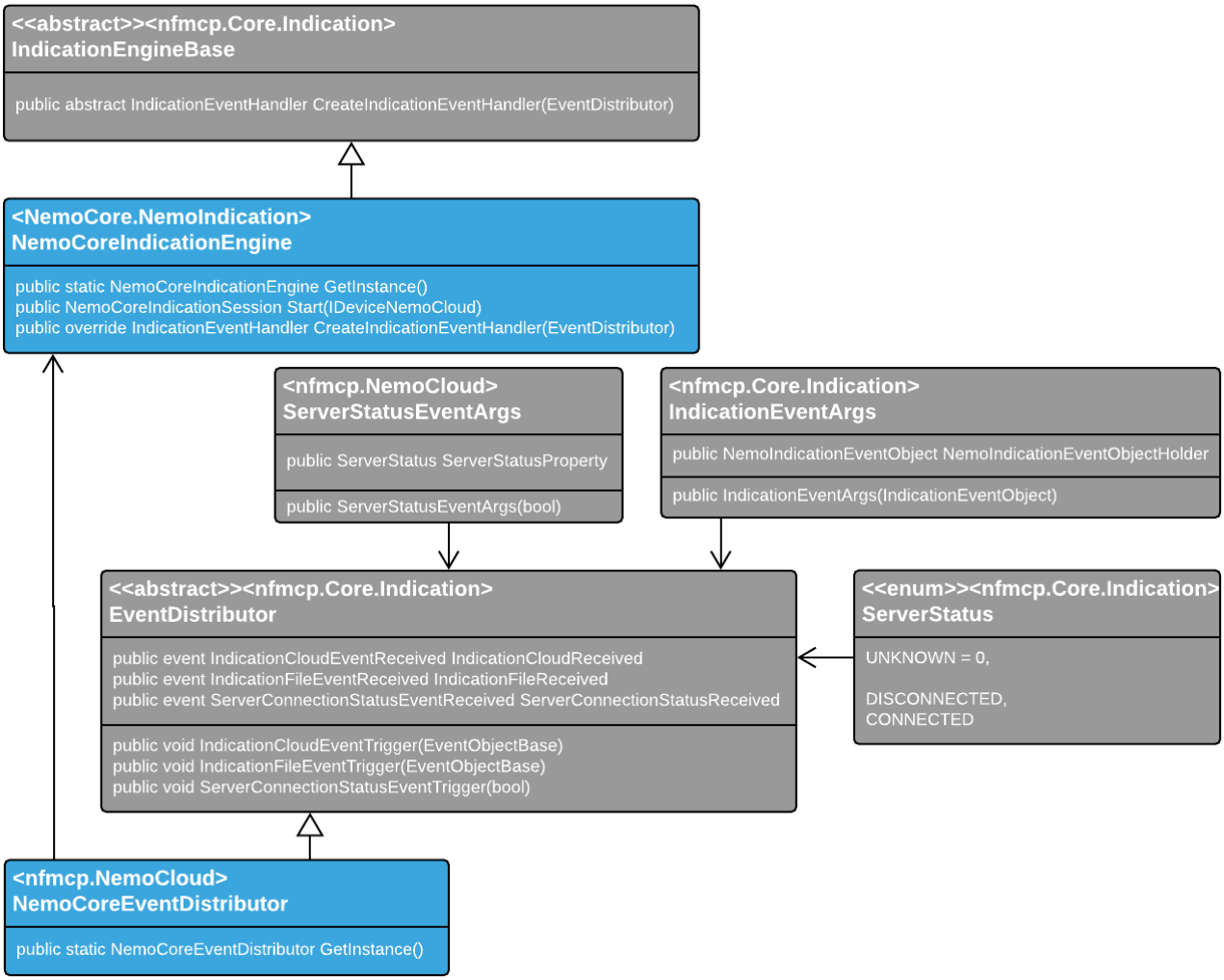

FIGURE 15. Class diagram from the event distributor

4.3 Event handler component

The event handler component in figure 16 is the caching system of the framework. Every session has its own event handler and the component's event queue thread is started in the session class and it is closed by calling the Close() method. If the event queue has data stored, the Close() method waits for the queue to be empty. The base class, EventHandlerBase, was already implemented to the NFM library and it was modified to suit the indication framework needs.

When the framework is operational, the incoming indication data goes through the event queue system but is instantly emptied from the queue if the server connection status is connected. If the server connection status is disconnected, the event queue holds the indication data. The event queue reacts dynamically to the server connection status and if the connection is regained, the queue system dumps out the stored data while still queueing the new data. The event queue is a FIFO buffer which means that the first item to go into the queue, also comes out first.

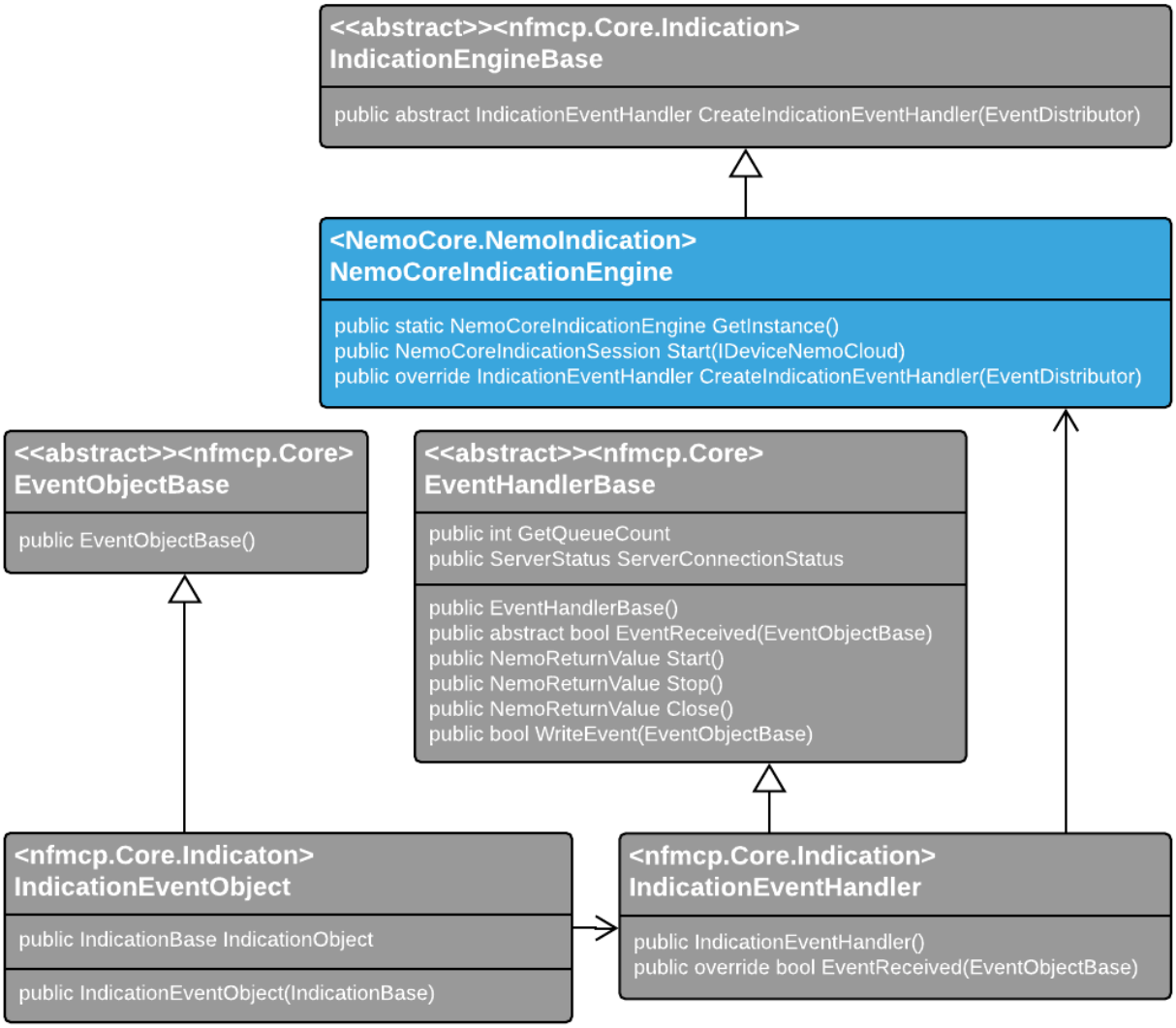

FIGURE 16. Class diagram from event handler

4.4 Indication handler and cloud component

The indication handler component holds the event listeners for the states of the server connection. When the class is initialized, it subscribes its event listeners to the two events mentioned in the EventDistributor class, namely the Indication-CloudReceived and IndicationFileReceived. The events start a new thread each time they pick up invokes and the maximum thread count is capped to 20 by using the ThreadPool class. In figure 17, we can see the structure of this component and how the classes relate to each other.

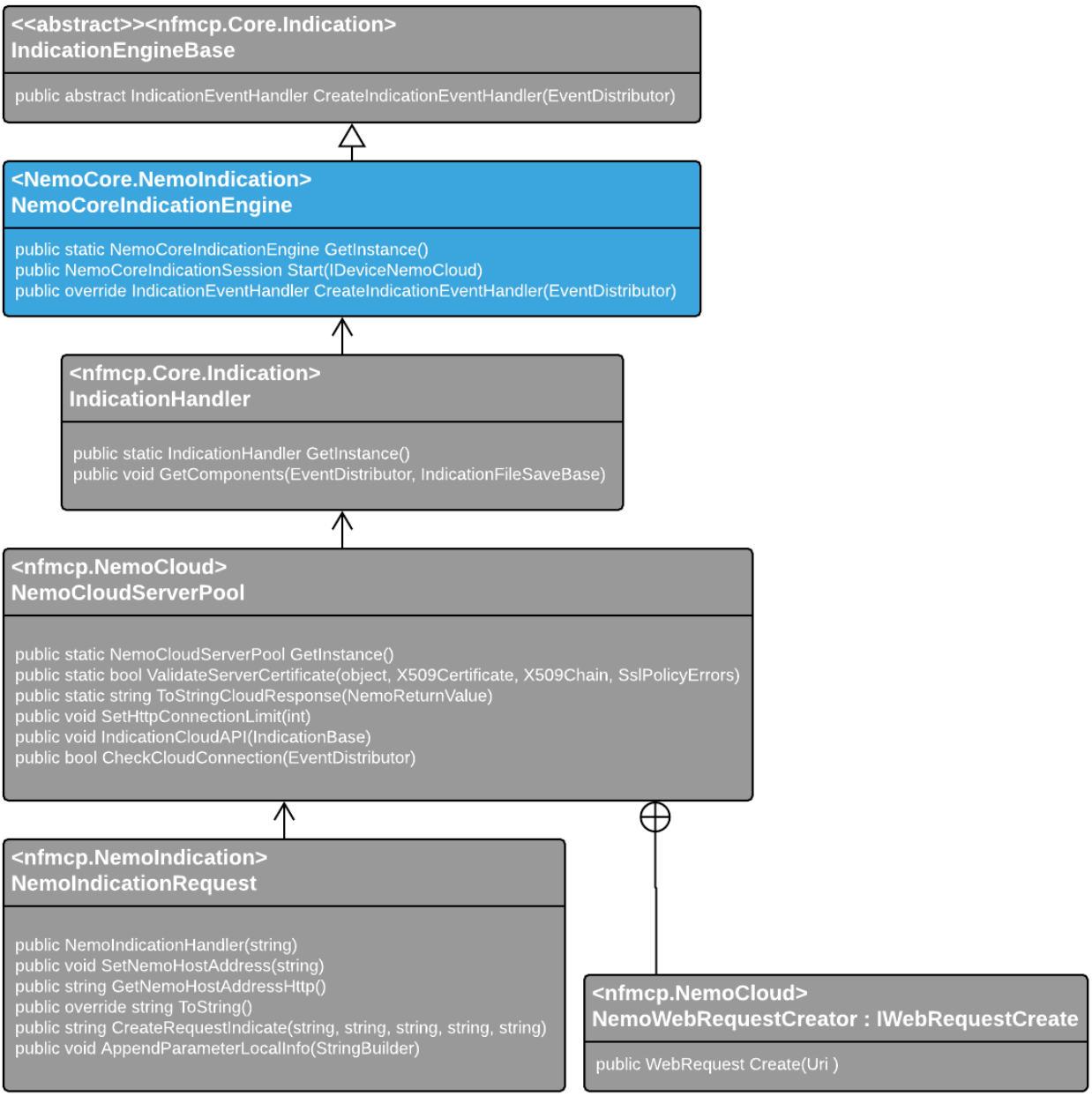

FIGURE 17. Class diagram from the indication handler and cloud implementation

The indications are sent to the server via HTTPS and the communication protocol is secured with TLS. The server connection functionality was mostly done using already functional NFM libraries which needed to be modified for indication purposes. Below, in figures 18 and 19, are the sequence diagrams of the event listeners in both server connection states.

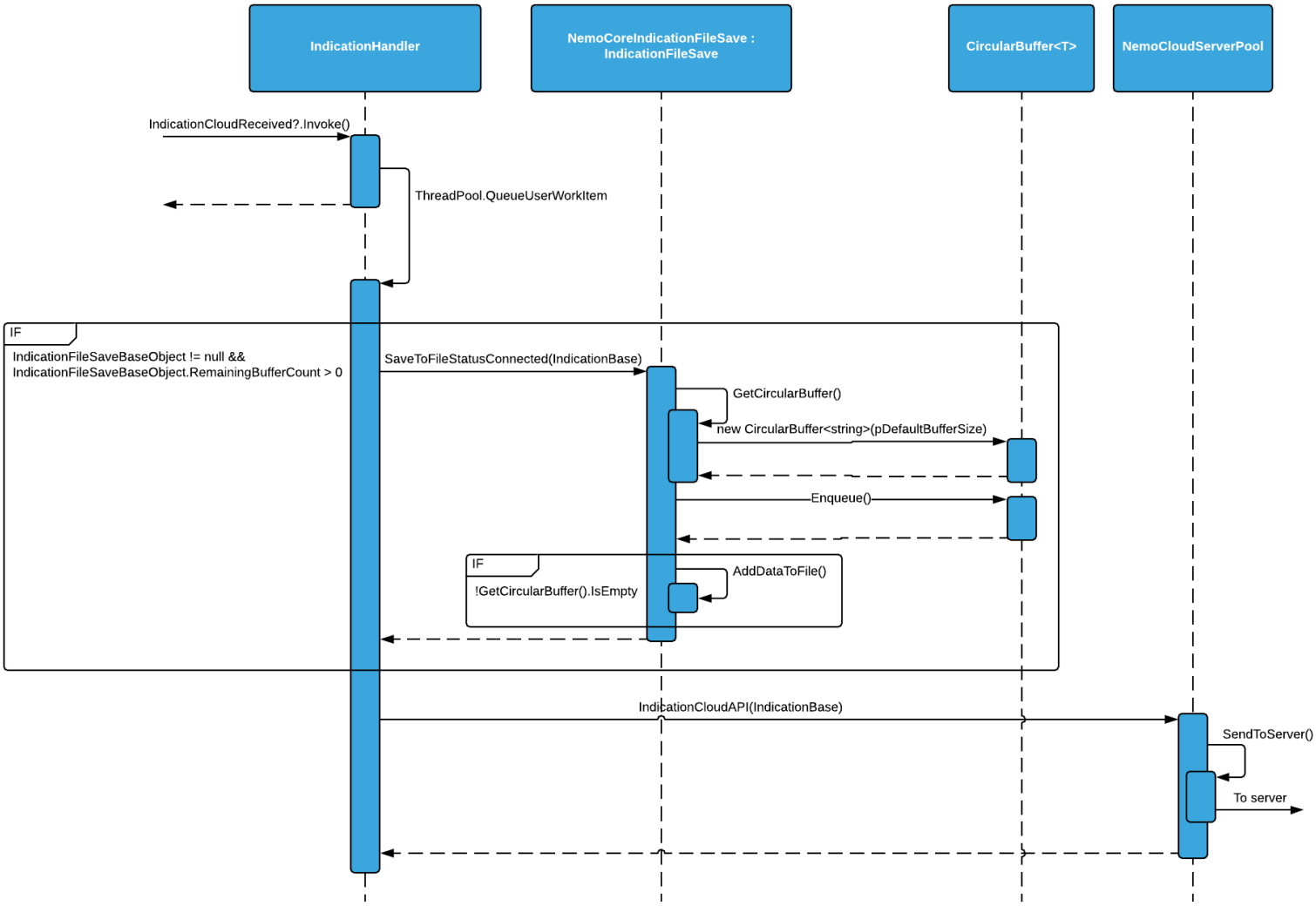

FIGURE 18. Sequence diagram from the IndicationHandler class when the server connection is in state connected

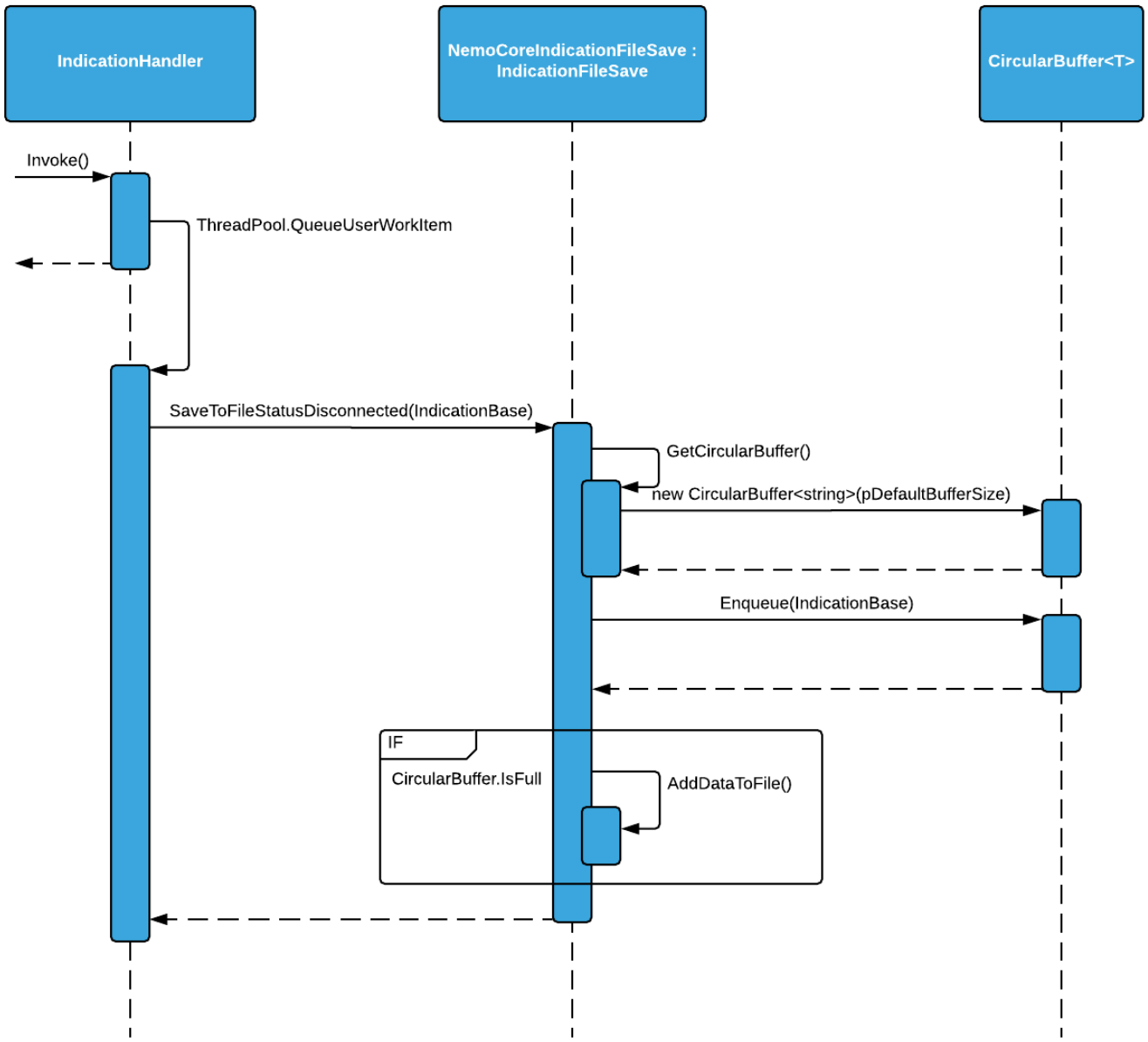

FIGURE 19. Sequence diagram from the IndicationHandler class when server connection is in state disconnected

4.5 File save component

The file save component creates the local log-files. The IndicationFileSaveBase class holds methods for both the server statuses, as shown in figure 20. If the server connection status is connected, the SavetoFileConnected() method empties the buffer, writing the data to file even before the buffer is full. While the server connection status is disconnected, the SavetoFileDisconnected() method saves the data to the file when the buffer is full.

The CircularBuffer class is a FIFO buffer and when the set buffer cap is reached, it dequeues the data into “List<string>” which is then written as Debuglog.fml file by using FileStream.Write(). The buffer also has a few properties for checking the item count in the buffer.

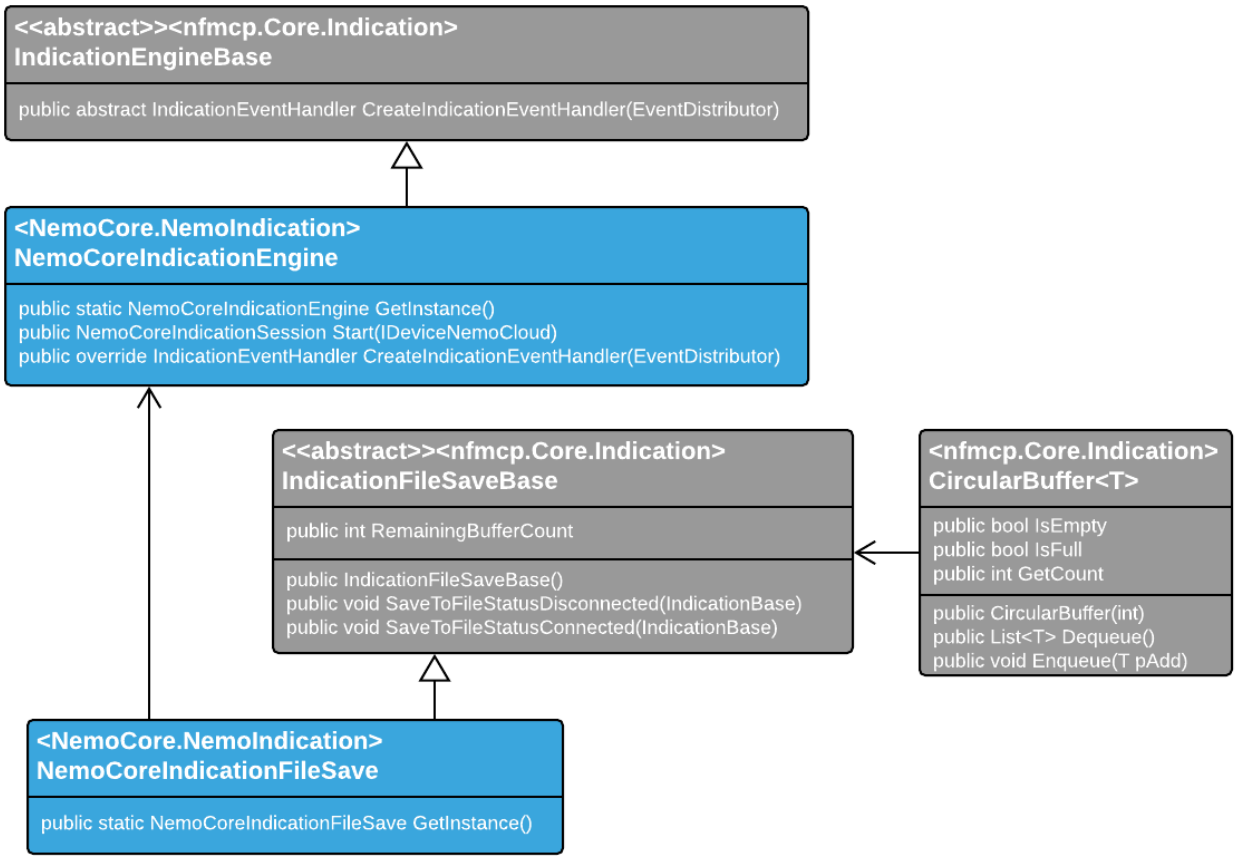

FIGURE 20. Class diagram from file saving implementation

5 TESTING THE FRAMEWORK

The development of this framework included testing the framework while it was programmed. To do this testing, I created a small console application as shown in figure 21. I implemented commands which would then run the wanted tests. This made the testing of the new features easy and fast.

5.1 Test data generator

Before testing the indication framework with actual data coming from the running NFM application, I created a random data generator to test the system. The generator pulled out random data from a predetermined pool of data. I made the generator produce new indications on certain intervals e.g. 500 milliseconds. This interval timer could be changed during the running of the program, to test how the framework could handle changes in the volume of data coming to the system.

Using the test generator, I was able to start multiple sessions. This meant that I could start e.g. four different generators and they would each produce indications with their own timer. This was used to simulate the amount of data that could come from using the NFM application with multiple terminals. This testing was particularly important when testing the key locations for thread locks such as semaphore, mutex and synchronization locks. The smart use of locks prevented the program from crashing due to the incoming data going through the framework.

With the test data generator, you could also simulate how the framework reacted to the server connection status changes. If the connection to the server is lost, the framework automatically starts to store data to the cache and to the local file. Furthermore, if the connection is regained, the framework will start dumping the data to the server from the cache. It would also empty its local data buffer as it writes the data to the local file.

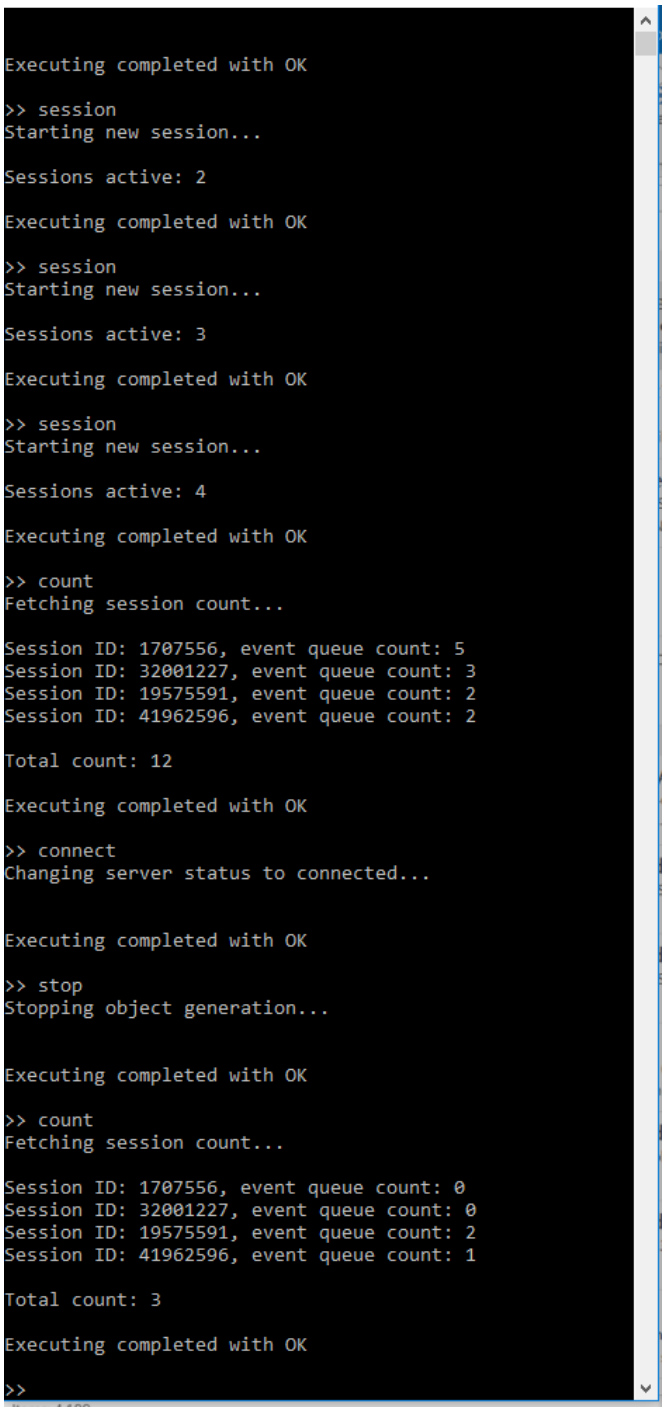

FIGURE 21. The test console of the framework. The test console was essential for the testing of the framework. Here we see some of the commands I made to test the framework

```
NFM Verbose: 0 : [09:14:52:368] ************************************************************************
NFM Verbose: 0 : [09:14:52:368] IndicationTestSession: OnTimedEvent...
NFM Verbose: 0 : [09:14:52:368] IndicationTestEngine: GetRandomPhonePreset...
NFM Verbose: 0 : [09:14:52:368] IndicationTestEngine: GetRandomPhonePreset completed
NFM Verbose: 0 : [09:14:52:368] ------------------------------------------------------
NFM Verbose: 0 : [09:14:52:368] IndicationTestSession: Date: 03/21/2018 09:14:52
NFM Verbose: 0 : [09:14:52:368] IndicationTestSession: OnTimedEvent: Random Object property values:
NFM Verbose: 0 : [09:14:52:368] IMEI: testDevice4
NFM Verbose: 0 : [09:14:52:368] IndicationString: This is device 4 indicaton string
NFM Verbose: 0 : [09:14:52:368] ------------------------------------------------------
NFM Verbose: 0 : [09:14:52:368] IndicationObjectFactoryBase: GetNemoIndicationEventObject...
NFM Verbose: 0 : [09:14:52:384] IndicationEventObject: Initialize...
NFM Verbose: 0 : [09:14:52:384] IndicationEventObject: Initialize completed
NFM Verbose: 0 : [09:14:52:384] IndicationObjectFactoryBase: GetNemoIndicationEventObject completed
NFM Verbose: 0 : [09:14:52:384] EventHandlerBase: WriteEvent: Queue count: 6
NFM Verbose: 0 : [09:14:52:415] IndicationTestSession SessionID: 1707556
NFM Verbose: 0 : [09:14:52:416] IndicationTestSession: OnTimedEvent completed
NFM Verbose: 0 : [09:14:52:417] IndicationEventHandler: EventReceived...
NFM Verbose: 0 : [09:14:52:443] IndicationEventHandler: EventReceived: Queue count: 6
NFM Verbose: 0 : [09:14:52:443] IndicationEventHandler: EventReceived: Server status: DISCONNECTED
NFM Verbose: 0 : [09:14:52:443] EventDistributor: IndicationFileEventTrigger...
NFM Verbose: 0 : [09:14:52:443] IndicationEventArgs: Initialize...
NFM Verbose: 0 : [09:14:52:443] IndicationEventArgs: Initialize completed
NFM Verbose: 0 : [09:14:52:443] IndicationHandler: Event_IndicationEventHandler_IndicationFileReceived...
NFM Verbose: 0 : [09:14:52:443] IndicationHandler: Event_IndicationEventHandler_IndicationFileReceived completed
NFM Verbose: 0 : [09:14:52:475] EventDistributor: IndicationFileEventTrigger completed
NFM Verbose: 0 : [09:14:52:475] IndicationEventHandler: EventReceived completed
NFM Verbose: 0 : [09:14:52:443] IndicationHandler: Event_IndicationEventHandler_IndicationFileReceived: StartNew...
NFM Verbose: 0 : [09:14:52:475] IndicationFileSaveBase: SaveToFileStatusDisconnected...
NFM Verbose: 0 : [09:14:52:475] IndicationFileSaveBase: GetCircularBuffer...
NFM Verbose: 0 : [09:14:52:475] IndicationFileSaveBase: GetCircularBuffer completed
NFM Verbose: 0 : [09:14:52:475] IndicationFileSaveBase: MarkBoundaries...
NFM Verbose: 0 : [09:14:52:475] IndicationFileSaveBase: MarkBoundaries completed
NFM Verbose: 0 : [09:14:52:475] CircularBuffer: Enqueue...
NFM Verbose: 0 : [09:14:52:475] CircularBuffer: NextPosition...
NFM Verbose: 0 : [09:14:52:490] CircularBuffer: NextPosition completed
NFM Verbose: 0 : [09:14:52:490] CircularBuffer: Enqueue completed
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: GetCircularBuffer...
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: GetCircularBuffer completed
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: GetCircularBuffer...
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: GetCircularBuffer completed
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: SaveToFileStatusDisconnected: CirularBuffer count: 6
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: GetCircularBuffer...
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: GetCircularBuffer completed
NFM Verbose: 0 : [09:14:52:490] IndicationFileSaveBase: SaveToFileStatusDisconnected completed
NFM Verbose: 0 : [09:14:52:490] IndicationHandler: Event_IndicationEventHandler_IndicationFileReceived: CircularBuffer count: 6
NFM Verbose: 0 : [09:14:52:506] IndicationHandler: Event_IndicationEventHandler_IndicationFileReceived: StartNew completed
NFM Verbose: 0 : [09:14:52:506] ************************************************************************
NFM Verbose: 0 : [09:14:52:850] IndicationTestEngine: IndicationObjectTicker...
NFM Verbose: 0 : [09:14:52:850] IndicationTestEngine: IndicationObjectTicker timer state: False
NFM Verbose: 0 : [09:14:52:850] IndicationTestEngine: IndicationObjectTicker completed
```

FIGURE 22. Trace logs of the framework's test version, showing the OnTimedEvent() method

In figure 22, the indication test generator has triggered the OnTimedEvent() method and you can see the process that every indication goes through. The IndicationEventHandler class is the caching system and the CircularBuffer class is the buffer which stores the data before writing it onto the local file. Every event listener (Event_IndicationEventHandler_IndicationFileReceived in traces above) method starts its own thread and the maximum thread count is capped to 20 by using the ThreadPool class.

You can also see that the server connection status is disconnected. This means that the indication data stays in the event queue which is the cache. When the connection status changes to connected, the event queue starts to slowly dump the data to the server and the data in the CircularBuffer is saved to the local file.

5.2 Further analysis

The sequence diagram in figure 23 shows the process of the OnTimedEvent() method which is part of the test generator.

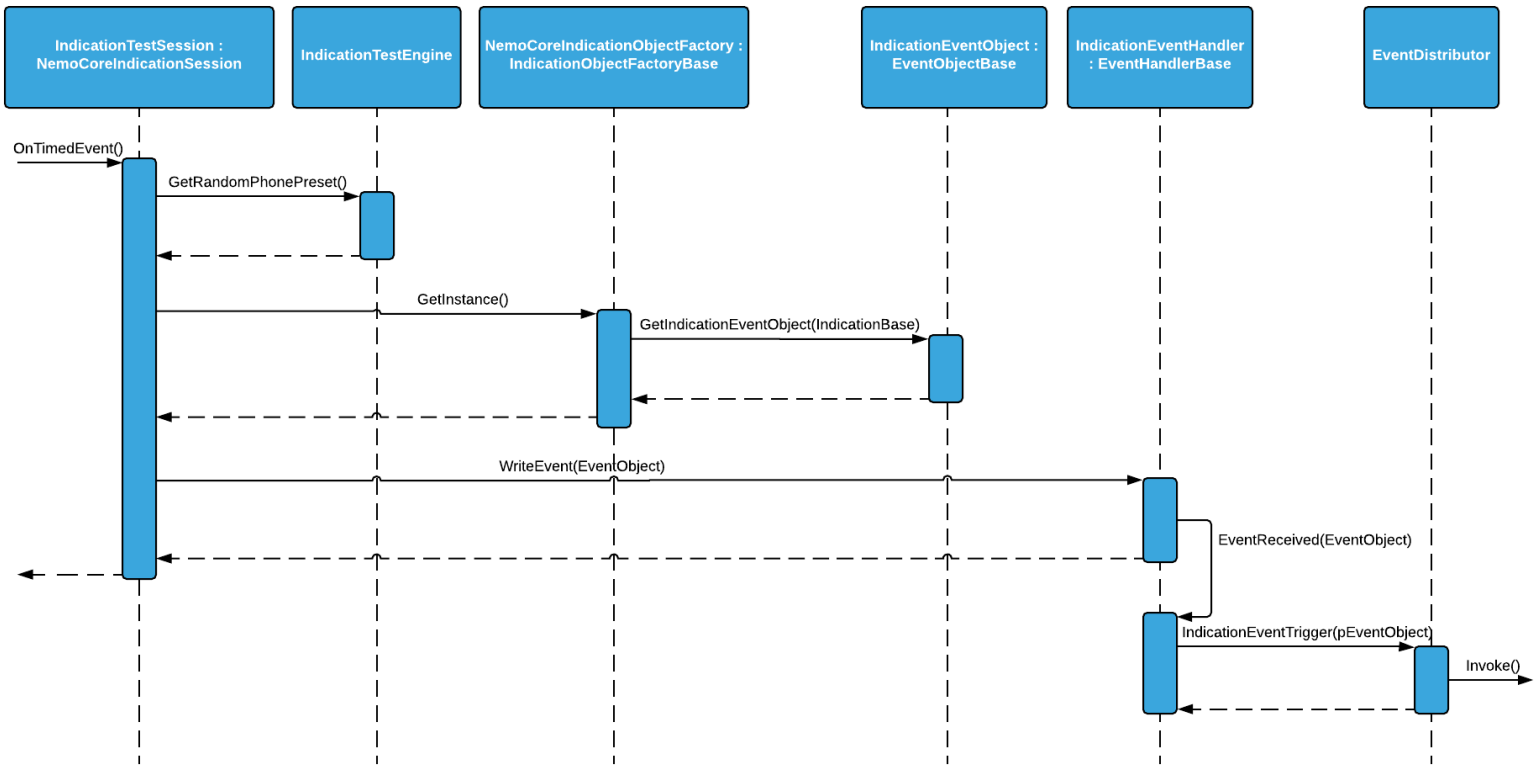

FIGURE 23. Sequence diagram from the test version of the framework.

By comparing the diagram in figure 23 to the trace logs in figure 22, you can see exactly how the indication object goes through the system to the EventDistributor class. In the EventDistributor class the method IndicationFileEventTrigger() is executed which triggers the event IndicationFileReceived. In the IndicationHandler class, the method Event_IndicationEventHandler_IndicationFileReceived() has been subscribed to this event as event listener and when the event signal comes, the listener executes its code. You can also see that the Event_IndicationEventHandler_IndicationFileReceived() method starts a new thread as the event listener method is completed before the indication has gone through StartNew traces.

6 AS PART OF THE NEMO FIRMWARE MANAGER

After getting the framework up and running with the console application in the test environment, I also tested it in real scenarios. I had two test terminals which I used to test the framework, as shown in figure 24. I added the framework's API to the DeviceThread class which has functionality involved with the "Check firmware for all devices" button and checked if I can retrieve the IMEI and the logged string I wanted. The tests were successful, and the framework worked as intended, as shown in figure 25.

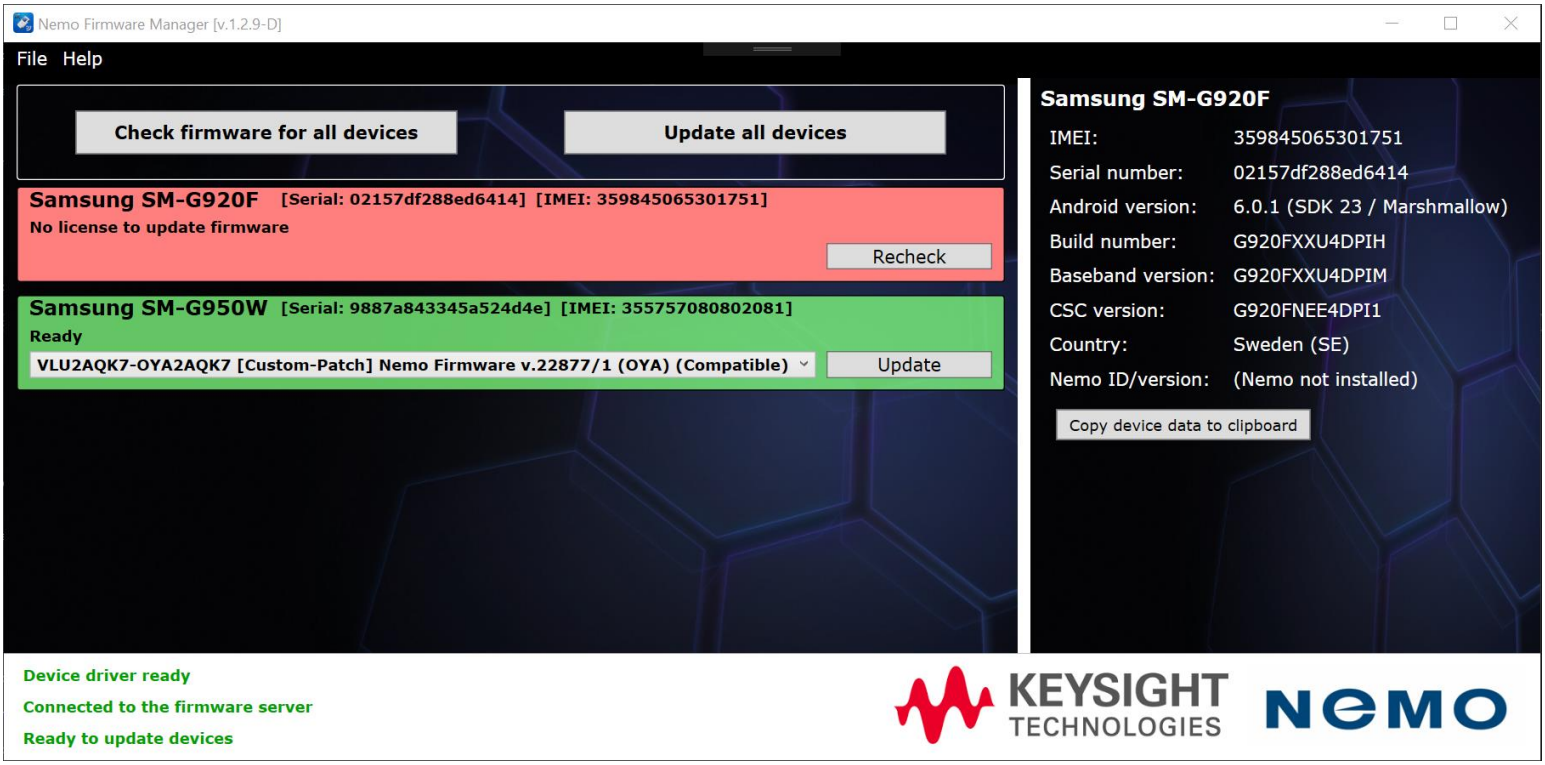

FIGURE 24. Nemo Firmware Manager Client GUI. Here the NFM has authenticated one terminal but the second terminal is showing license error

By clicking the "Check firmware for all devices" button, the NFM also utilizes the indication framework which will save some trace log data to demonstrate the framework's functionality as part of the NFM process.

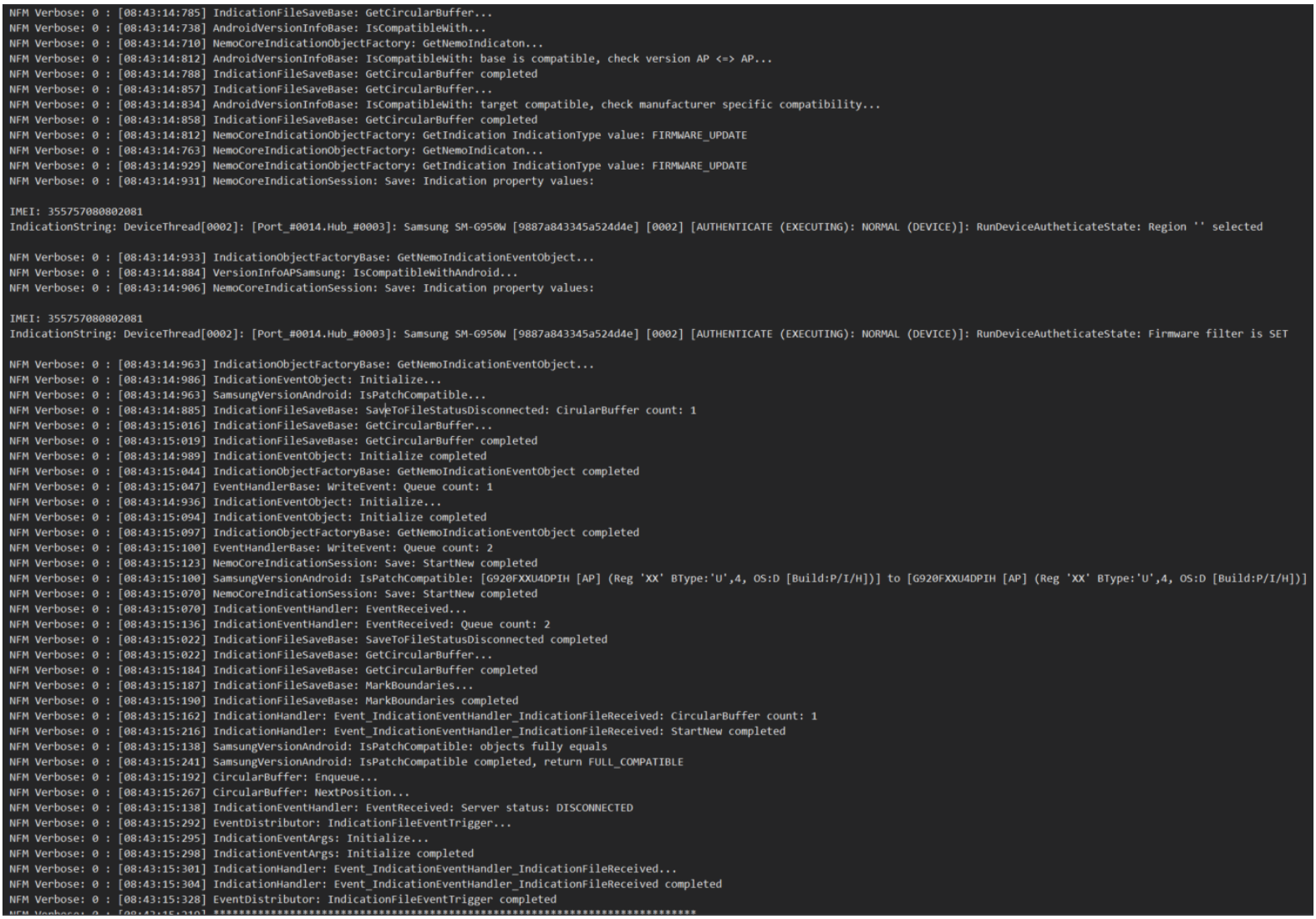

FIGURE 25. Framework logging event data in the server connection state disconnected, as part of the Nemo Firmware Manager

After testing the framework with the server connection in the disconnected state, I also tested it with the status set to connected. By comparing the figures 25 and 26, we can see that the indicaton data has gone through the framework and has been successfully sent to the server. This is confirmed in figure 26 which is the NFM admin GUI for the logged indication data.

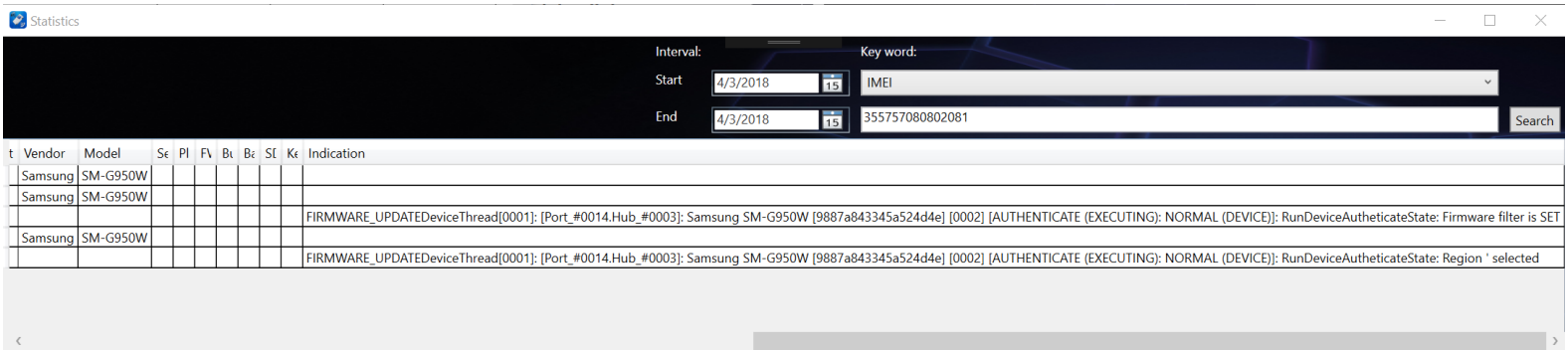

FIGURE 26. Nemo Firmware Manager admin GUI for the logged data

7 CONCLUSION

The goal of this project was to make modular and versatile logging system for the Nemo Firmware Manager. Important points were that it could be easily modified and that implementing new logging methods to the framework would not be hindered by its architecture design. The criteria were met with success and the results were excellent. The framework has plenty of potential for further development and it will be continued to be developed in the future.

The original idea was to improve the NFM GUI for checking the indication data logged by the framework. As the time went on, the boundaries of the project had to be rescoped. However, it is still possible to benefit from the framework as the old GUI is still in place and has the basic functionalities e.g. searching indication data with IMEI.

In term of further development, decisions must be made regarding what events will be logged from the NFM with the indication framework and code the case-specific logic to each indication type.

During this project, I learned a great deal about OOP and OOAD; asynchronous programming; documentation; design patterns; how to effectively utilize libraries which have been implemented by others; version control; and how to iterate through a project using agile software development. These are all valuable skills in software development and Keysight Technologies has enabled me to practice these skills while working for them on this thesis.

REFERENCES

1. Hungarian Notation. 2017. Computer Hope. Date of retrieval 29.03.2018. Available: https://www.computerhope.com/jargon/h/hungarian-notation.htm

2. Design Patterns. 2018. Sourcemaking. Date of retrieval 14.03.2018. Available: https://sourcemaking.com/design_patterns

3. Factory Method Design Pattern. 2018. Sourcemaking. Date of retrieval 20.03.2018. Available: https://sourcemaking.com/design_patterns/factory_method

4. Purdy, Doug 2002. Exploring the Factory Design Pattern. Date of retrieval 18.02.2018. Available: https://msdn.microsoft.com/en-us/library/ee817667.aspx

5. Singleton Design Pattern. 2018. Sourcemaking. Date of retrieval 20.03.2018. Available: https://sourcemaking.com/design_patterns/singleton

6. Brown, Simon 2018. C4 Modeling. Date of retrieval 08.03.2018. Available: https://c4model.com/

7. Thread Synchronization. 2015. Microsoft Docs. Date of retrieval 28.03.2018. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/thread-synchronization

8. Standard .NET event patterns. 2016. Microsoft. Date of retrieval 28.03.2018. Available: https://docs.microsoft.com/en-us/dotnet/csharp/event-pattern